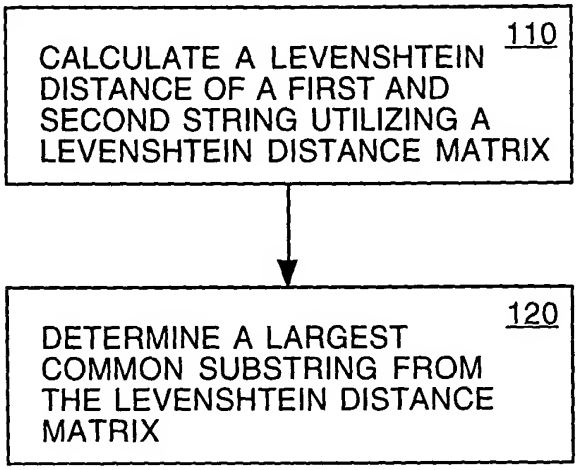
110
CALCULATE A LEVENSHTEIN DISTANCE OF A FIRST AND SECOND STRING UTILIZING A LEVENSHTEIN DISTANCE MATRIX
120
DETERMINE A LARGEST COMMON SUBSTRING FROM THE LEVENSHTEIN DISTANCE MATRIX

FIGURE 1

		W	I	D	G	E	T		I	N	C	230
	0	1	2	3	4	5	6	7	8	9	10	240
A	1	1	2	3	4	5	6	7	8	9	10	
C	2	2	2	3	4	5	6	7	8	9	9	
M	3	3	3	3	4	5	6	7	8	9	10	
E	4	4	4	4	4	4	5	6	7	8	9	
	5	5	5	5	5	5	5	5	6	7	8	
W	6	5	6	6	6	6	6	6	6	7	8	
I	7	6	5	6	7	7	7	7	6	7	8	
D	8	7	6	5	6	7	8	8	7	7	8	
G	9	8	7	6	5	6	7	8	8	8	8	
E	10	9	8	7	6	5	6	7	8	9	9	
T	11	10	9	8	7	6	5	6	7	8	9	
	12	11	10	9	8	7	6	5	6	7	8	
C	13	12	11	10	9	8	7	6	6	7	7	
O	14	13	12	11	10	9	8	7	7	7	8	250

210 220

FIGURE 2A

		W	I	D	G	E	T		I	N	C
	0	1	2	3	4	5	6	7	8	9	10
A	1	1	2	3	4	5	6	7	8	9	10
C	2	2	2	3	4	5	6	7	8	9	9
M	3	3	3	3	4	5	6	7	8	9	10
E	4	4	4	4	4	4	5	6	7	8	9
	5	5	5	5	5	5	5	5	6	7	8
W	6	5	6	6	6	6	6	6	6	7	8
I	7	6	5	6	7	7	7	7	6	7	8
D	8	7	6	5	6	7	8	8	7	7	8
G	9	8	7	6	5	6	7	8	8	8	8
E	10	9	8	7	6	5	6	7	8	9	9
T	11	10	9	8	7	6	5	6	7	8	9
	12	11	10	9	8	7	6	5	6	7	8
C	13	12	11	10	9	8	7	6	6	7	7
O	14	13	12	11	10	9	8	7	7	7	8

260

FIGURE 2B

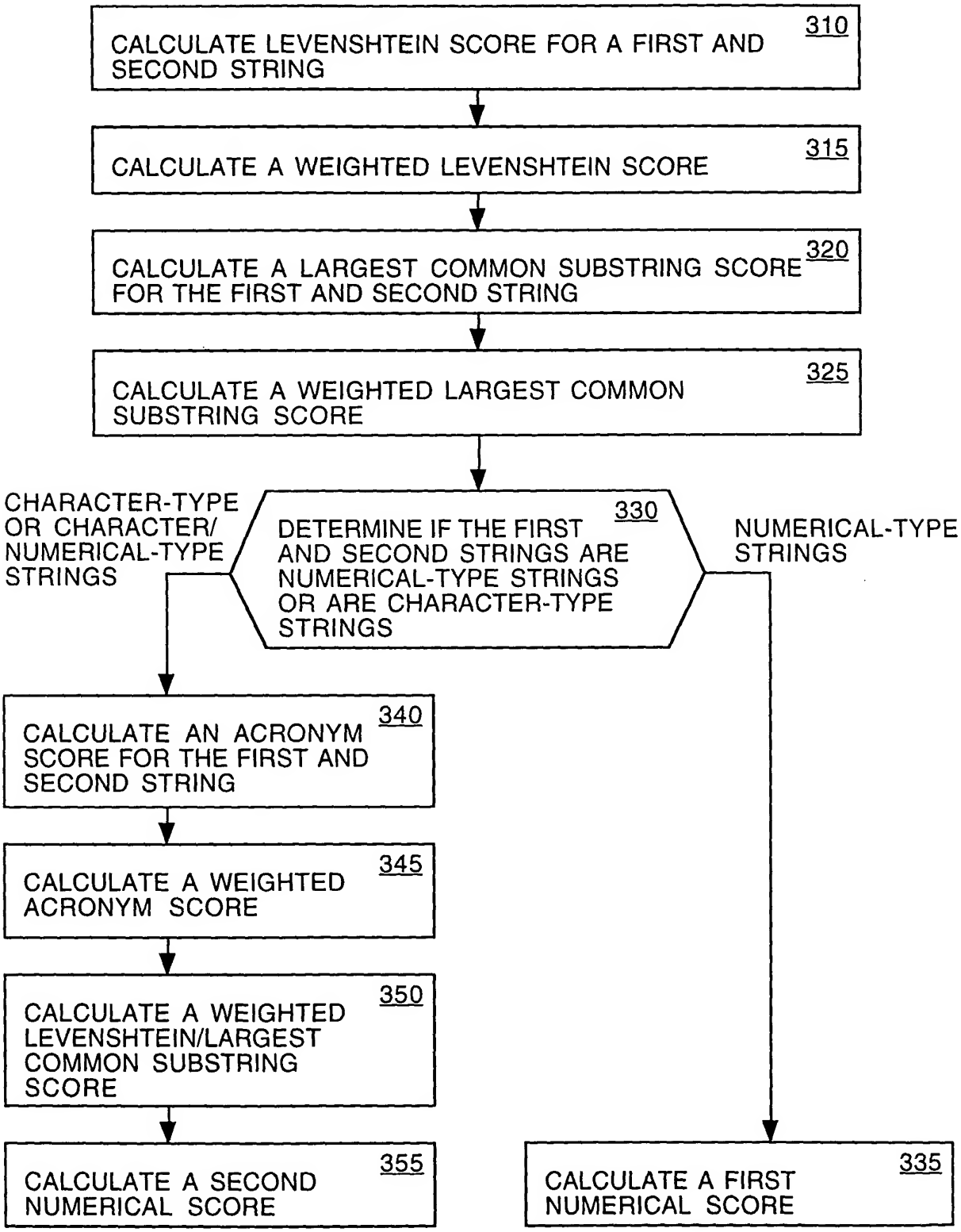
CALCULATE LEVENSHTEIN SCORE FOR A FIRST AND SECOND STRING
310
CALCULATE A WEIGHTED LEVENSHTEIN SCORE
315
CALCULATE A LARGEST COMMON SUBSTRING SCORE FOR THE FIRST AND SECOND STRING
320
CALCULATE A WEIGHTED LARGEST COMMON SUBSTRING SCORE
325
DETERMINE IF THE FIRST AND SECOND STRINGS ARE NUMERICAL-TYPE STRINGS OR ARE CHARACTER-TYPE STRINGS
330
CHARACTER-TYPE OR CHARACTER/ NUMERICAL-TYPE STRINGS
NUMERICAL-TYPE STRINGS
CALCULATE AN ACRONYM SCORE FOR THE FIRST AND SECOND STRING
340
CALCULATE A WEIGHTED ACRONYM SCORE
345
CALCULATE A WEIGHTED LEVENSHTEIN/LARGEST COMMON SUBSTRING SCORE
350
CALCULATE A SECOND NUMERICAL SCORE
355
CALCULATE A FIRST NUMERICAL SCORE
335

FIGURE 3

```
/**
* This function computes the Levenshtein distance of two strings
   *
* @param s the first string
* @param t the second string
* @param n length of s
* @param m length of t
* @return Levenshtein distance of s and t
*/
public static int levenshteinComputations(String s,
                                          String t,
                                          int n,
                                          int m) {

    int i;                  // iterates through s
    int j;                  // iterates through t
    int k;                  // used to initialize s and t
    char s_i;               // ith character of s
    int jNext;
    int iNext;
    int prevJVal;
    int startAt = 0;
    int a, b, c, temp;
    boolean substringBroken = false;

    // Levenshtein matrix
    //int[][] levenshteinDist = returnInitMatrix(n+1, m+1);

    // Step 1 (takes care of a null string)
    if ((n == 0) || (m == 0)) {
       levenshteinDist[n][m] = (n == 0)?m:n;
       return 0;
    }

    // Step 1.5 (eliminate the common initial string in the strings)
    temp = (m < n)?m:n;
    for (i = 0; (i < temp) && !substringBroken; i++) {
       if (s.charAt(i) != t.charAt(i)) {
          startAt = i;
          substringBroken = true;
       }
    }

    if (!substringBroken && i == temp) {
       startAt = temp;
       // Return immediately if one string is completely contained in
       // the other.
       levenshteinDist[n][m] = (m > n)?(m - n):(n - m);
       return startAt;
    }
```

FIGURE 4A

```
    // Step 2 (initialize the elements of the matrix)
    for (k = 0, i = startAt; i <= n; i++)
       levenshteinDist[i][startAt] = k++;

    for (k = 0, j = startAt; j < m; j++) {
       levenshteinDist[startAt][j] = k++;
       tBuffer[j] = t.charAt(j);
    }
    levenshteinDist[startAt][m] = k;

    // Step 3 (perform the computation)
    for (i = startAt; i < n;) {
       s_i = s.charAt(i);
       iNext = i+1;

       // optimization: minimize array references by setting the
       // temporary variable prevJVal. Set the previous j value to the
       // value at levenshtein[iNext][startAt] in order to initialize it.
       // (see initialization at Step 2)
       prevJVal = levenshteinDist[iNext][startAt];

       // Step 4
       for (j = startAt; j < m;) {
          jNext = j+1;
          a = levenshteinDist[i][jNext]+1;

          // b = levenshteinDist[iNext][j]+1;
          b = prevJVal+1;

          // Step 5
          c = (s_i == tBuffer[j])?levenshteinDist[i][j]:levenshteinDist[i][j]+1;

          // Step 6
          temp = (a < b)?a:b;
          levenshteinDist[iNext][jNext] = prevJVal = (c < temp)?c:temp;
          j = jNext;
       }

       i = iNext;
    }

    // Step 7 (return the levenshtein matrix and the starting position)

    return startAt;
}
```

FIGURE 4B

```
/**
* This function computes the largest common substring score of two strings
*
* @param s the first string
* @param t the second string
*/

prevString1 = s;
prevString2 = t;

previousLevScore = 1 - (((float) (distance << 1))/
                ((float) (lengthS + lengthT)));

returnVal += previousLevScore;

if (!largeLengthDiff) {

  // Calculate the substring score if this is the case
  if (returnVal > 0 && returnVal < maxScore) {
    int currMaxLength = 0;
    int k, l;
    int currIteration;
    int d1;
    int d1_length;
    int d2;

    // Loop through the rows, then the columns
    for (k = startAt; k <= lengthS; k++) {
      currIteration = k;
      d1 = 0;
      d1_length = 0;

      for (l = startAt;
          (l <= lengthT) && (currIteration <= lengthS);
          l++) {
          d2 = levenshteinDist[currIteration][l];

          if (d1 == d2)
              d1_length++;
          else
              d1_length = 1;

          d1 = d2;

          currIteration++;
      }

      if (d1_length > currMaxLength)
          currMaxLength = d1_length;
    }
```

FIGURE 4C

```
// Loop through the columns then the rows
for (l = startAt; l <= lengthT; l++) {
  currIteration = l;
   d1 = 0;
   d1_length = 0;

  for (k = startAt;
     (k <= lengthS) && (currIteration <= lengthT);
     k++) {
     d2 = levenshteinDist[k][currIteration];

     if (d1 == d2)
        d1_length++;
     else
        d1_length = 1;

     d1 = d2;

     currIteration++;
  }
   if (d1_length > currMaxLength)
     currMaxLength = d1_length;
}

// Make sure that the matching substring is not the
// initial match
 if (startAt > currMaxLength) {
   currMaxLength = startAt;
 } else {
   currMaxLength--;
}
```

FIGURE 4D

```
/**
 * Takes two strings and gets a score based on their acronyms
 *
 * @param str1 first string
 * @param str2 second string
 * @param m the multipliers to return
 * @return the score of comparison between the acronyms
 */
public static float scoreAcronyms(String str1,
                                  String str2,
                                  float partialMatch,
                                  float exactMatch) {
  int acr1Length = 1;
  int acr2Length = 1;
  int str1Length = str1.length();
  int str2Length = str2.length();
  int minLength, i;

  if (str1 == null || str2 == null) {
    return 0;
  }

  // get the acronym representation of string 1
  acr1[0] = str1.charAt(0);
  for (i = 1; i < str1Length; i++) {
    if (str1.charAt(i) == ' ' && (++i) < str1Length)
      acr1[acr1Length++] = str1.charAt(i);
  }

  // if there is only one word, copy the entire string into the acronym
  if (acr1Length == 1) {
    for (i = 1; i < str1Length; i++) {
      acr1[acr1Length++] = str1.charAt(i);
    }
  }

  // get the acronym representation of string 2
  acr2[0] = str2.charAt(0);
  for (i = 1; i < str2Length; i++) {
    if (str2.charAt(i) == ' ' && (++i) < str2Length)
      acr2[acr2Length++] = str2.charAt(i);
  }

  // if there is only one word, copy the entire string into the acronym
  // this allows us to match already-acronymized names to non-acronymized
  // strings (e.g., ge = general electric)
  if (acr2Length == 1) {
    for (i = 1; i < str2Length; i++) {
      acr2[acr2Length++] = str2.charAt(i);
    }
  }
```

FIGURE 4E

```
    // see how equal the acronyms are.
    minLength = (acr1Length > acr2Length)?acr2Length:acr1Length;
    for (i = 0; (i < minLength) && (acr1[i] == acr2[i]); i++) {}


    // give the acronyms a non-zero score only if the loop above completed.
    if (i == minLength)
      return (acr1Length == acr2Length)?exactMatch:partialMatch;
    else
      return 0;
}
```

FIGURE 4F

```
/**
* Returns a consolidated Levenshtein, Substring, and Acronymn score
*
* @param s the first string
* @param t the second string
* @param maxScore the maximum allowable returnable score
* @param m the multipliers to use
* @return the consolidated score of both these substrings
*/
 public static float consolidatedScore(String s,
                                       String t,
                                       float maxScore) {
  float returnVal = 0;

  // If the strings are equal, we are done, just return the max score
  if (s.equals(t))
     return maxScore;
  else {
     // Previous strings have been cached to save computations
     if (prevString1.equals(s) && prevString2.equals(t)) {
       returnVal = maxScore;
     } else {
        // Set the lengths we're going to use for computations.
       int distance;
        int lengthS = s.length();
        int lengthT = t.length();
        int longerLength = (lengthS > lengthT)?lengthS:lengthT;
        int shorterLength = (lengthS > lengthT)?lengthT:lengthS;
       int startAt;
        boolean largeLengthDiff;

        // If the lowest among top 25 scores is less than 0, then see
        // if we can just approximate the levenshtein distance
        if (largeLengthDiff = (longerLength > (shorterLength << 2))) {
           distance = longerLength - shorterLength + 1;
           startAt = 0;
        } else {
           startAt = levenshteinComputations(s,
                                             t,
                                             lengthS,
                                             lengthT);

           distance = levenshteinDist[lengthS][lengthT];
        }
```

FIGURE 4G

```
                // Compute and scale the substring score, add it to the
                // returned value.
                returnVal += (((float) currMaxLength) /
                        ((float) ((lengthS > lengthT)?lengthS:lengthT)));

            }
        }
    }
    // Scale down because both Levenshtein and Substring are out of one
    returnVal *= 0.5;
    return returnVal;
  }
}
```

FIGURE 4H

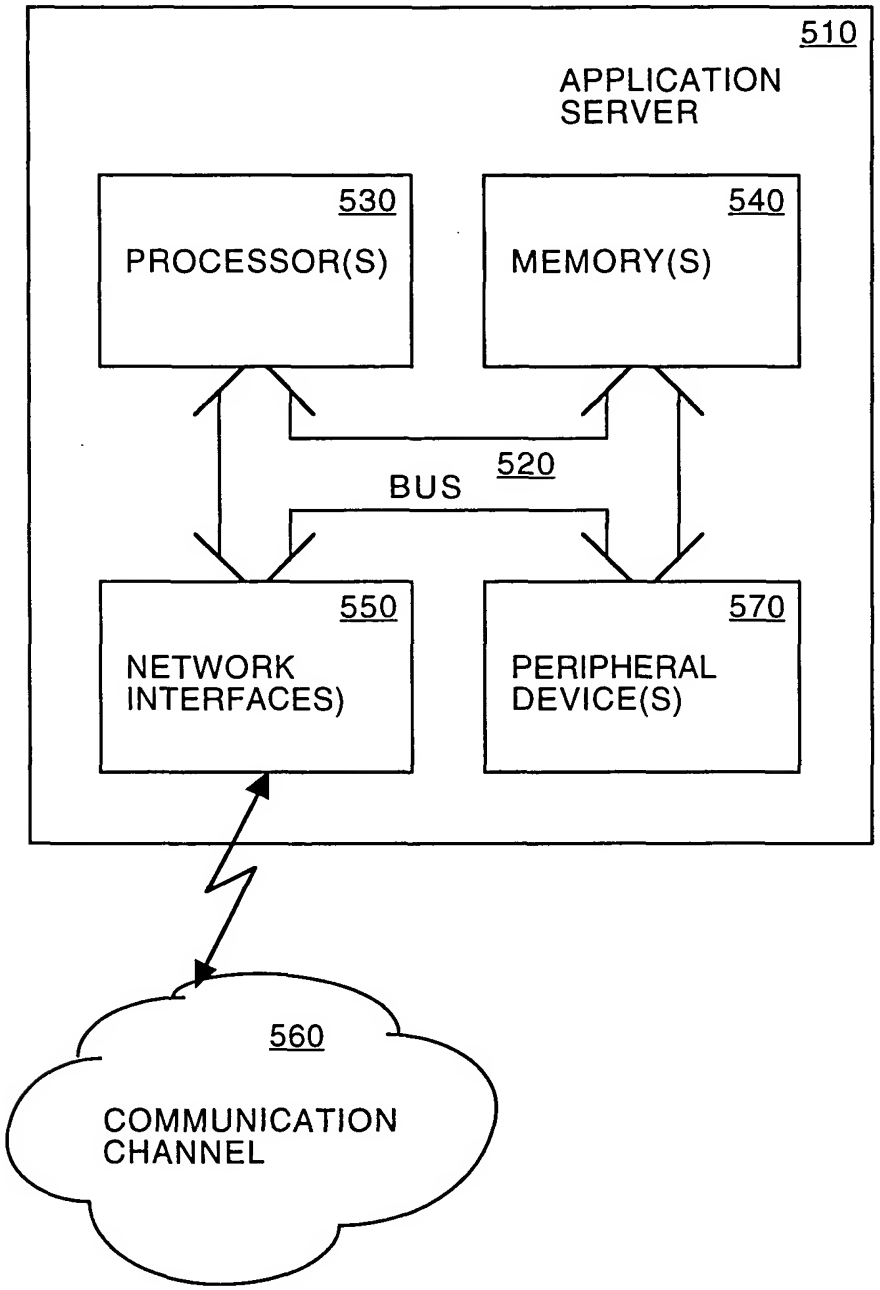
510
APPLICATION SERVER
530
PROCESSOR(S)
540
MEMORY(S)
520
BUS
550
NETWORK INTERFACES)
570
PERIPHERAL DEVICE(S)
560
COMMUNICATION CHANNEL

FIGURE 5